Défis 2025

Groupe de Recherche CNRS - Génie de la Programmation et du Logiciel

Philippe Collet[1], Lydie Du Bousquet[2], Laurence Duchien[3], Pierre-Etienne Moreau[4]

1. Laboratoire I3S, Université Nice-Sophia Antipolis
Campus SophiaTech, 930 Route des Colles
BP 145 06903 Sophia Antipolis Cedex, France
philippe.collet@unice.fr

2. Laboratoire LIG, Université Joseph Fourier
BP 72 - 38402 Saint Martin d'Hères – France
Lydie.Du-Bousquet@imag.fr

3. Laboratoire CRIStAL, Université Lille 1 Sciences et Technologies
Cité Scientifique – Bat M3 – 59655 Villeneuve d'Ascq Cedex, France
laurence.duchien@univ-lille1.fr

4. Laboratoire LORIA, Université de Lorraine
615, rue du jardin botanique, CS 20101, 54603 Villers-lès-Nancy Cedex, France
Pierre-Etienne.Moreau@loria.fr

RÉSUMÉ De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les dix ans à venir, que ce soit pour faciliter la vie des concepteurs et mainteneurs de systèmes informatiques, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique.
Ce texte résume les travaux menés sur les défis du Génie de la Programmation et du Logiciel à l'horizon 2025. Ces travaux ont été l'occasion de présentations et d'échanges lors des journées nationales du Groupe de Recherche Génie de la Programmation et du Logiciel en juin 2014 et lors d'une journée en septembre 2014 à Paris.

ABSTRACT. New paradigms, languages, modeling, verification, testing approaches and new tools in the field of programming and software should be created in the next 10 years, whether to make life easier for designers and maintainers of computer systems, to model and reliable software or to anticipate technological change.
This text summarizes the challenges in the Programming and Software Engineering field on the horizon 2025. This work has been presented and discussed during the national days of the

Technique et science informatiques – n° 3/2015, 311-324

Research Group on Programming and Software Engineering in June 2014 and in September 2014 in Paris.

MOTS-CLÉS : génie logiciel, programmation, langage, exigences, spécification, compilation, vérification, validation, test, exécution, cycle de vie, fiabilité, robustesse, énergie, pilote de périphérique, adaptation.

KEYWORDS: software engineering, programming, language, requirements, specification, compilation, verification, validation, test, runtime, lifecycle, reliability, strength, energy, device driver, adaptation.

DOI:10.3166/TSI.34.293-306 © Lavoisier 2015

1. Introduction

Le Groupe de Recherche en Génie de la Programmation et du Logiciel (GdR GPL) a lancé en janvier 2014 un appel à défis 2025, qui fait suite à l'appel à défis 2020 lancé en 2010. Les défis 2010 ont été publiés dans TSI en 2012 (Duchien et Ledru, 2012). Nous reprenons dans ce document les réponses de l'appel 2014, tout en le complétant par d'autres travaux de façon à aborder l'ensemble des thèmes de la communauté GPL.

Les chercheurs du GdR GPL travaillent sur des abstractions logicielles et les fondements scientifiques associés, en vue de proposer des solutions bien fondées et pratiques permettant la production et la maintenance de logiciels de qualité. Ces abstractions et fondements interviennent à toutes les étapes du cycle de vie du logiciel, de sa conception à son exploitation et à sa maintenance, et prennent différentes formes dans des théories, techniques, outils et méthodes de modélisation, de validation, de vérification, dans les langages de programmation et dans les plates-formes d'exécution. Quelle que soit l'étape du cycle de vie étudiée, l'objectif est d'offrir des moyens de construction de logiciel satisfaisant les besoins exprimés tout en apportant une aide aux concepteurs et en maîtrisant coûts et délais.

Les problématiques sur lesquelles les chercheurs se positionnent sont renouvelées par de nouveaux domaines d'application dans tous les secteurs (informatique embarquée, intelligence ambiante, informatique dématérialisée), de nouveaux enjeux de société (développement durable, économies d'énergie, objets connectés), ainsi que par la diversité des fonctionnalités à fournir et à adapter aux besoins spécifiques de clients et aux modifications continuelles.

Les questions auxquelles les chercheurs du GdR GPL tentent de répondre sont :

– Comment concevoir et réaliser des systèmes qui soient, par exemple, personnalisés ou capables de s'adapter par eux-mêmes aux changements de leur environnement, pouvant facilement découvrir, sélectionner et intégrer des services disponibles à l'exécution ?

– Comment construire des langages, compilateurs, pilotes et supports d'exécution qui prennent en compte les nouvelles architectures, multiples, hétérogènes tout en maîtrisant les dépenses d'énergie ?

– Comment valider et vérifier ces logiciels qui vont évoluer dans des environnements de plus en plus imprévisibles ?

Ce document reprend ces questions et apporte un éclairage sous forme de défis en suivant les différentes étapes du cycle de vie du logiciel. La section 2 reprend les défis relevant des exigences à la réalisation des logiciels. La section 3 enchaîne sur les défis de la compilation jusqu'à l'exécution des logiciels. La section 4 présente les défis nouveaux concernant les méthodes et outils de validation et de vérification. La section 5 conclut ce document. Finalement, les défis rédigés par les chercheurs de la communauté du GdR GPL en 2014 sont ajoutés en annexe à ce document.

2. Des exigences à la réalisation des logiciels

2.1. Modéliser et analyser des systèmes de systèmes

La complexité croissante de notre environnement socio-économique produit de très grands systèmes, le plus souvent concurrents, distribués à grande échelle et parfois composés d'autres systèmes. Cela engendre de nouveaux défis de l'expression des exigences jusqu'à la réalisation des systèmes. On appelle système de systèmes (SdS) le résultat de l'intégration de plusieurs systèmes indépendants et interopérables, interconnectés dans le but de faire émerger de nouvelles fonctionnalités.

La complexité d'un SdS réside dans cinq caractéristiques intrinsèques qui sont : l'indépendance opérationnelle des systèmes constituants, l'indépendance managériale de ces mêmes systèmes, la distribution géographique, l'existence de comportements émergents (souhaités ou non), et enfin un processus de développement évolutionnaire.

La conception de SdS présente des différences importantes par rapport à la conception de systèmes classiques. Elle intègre les systèmes constituants existants, parfois traités comme des systèmes hérités (en anglais *legacy*), c'est-à-dire des boîtes noires. Il faut trouver un équilibre entre les exigences émergentes du SdS et celles préexistantes des systèmes intégrés, parfois contradictoires en terme de performances, sécurité ou encore tolérance aux pannes. Le cycle de vie d'un SdS peut être long. Il faut compter des dizaines d'années pour des SdS tels que ceux du domaine de la défense, par exemple. Au cours d'un cycle, le SdS n'est pas figé. A tout moment, de nouveaux systèmes peuvent être ajoutés ou supprimés, à la conception comme à l'exécution. De plus, chaque système constituant peut subir des évolutions, des mises à jour, qui peuvent avoir une incidence sur l'ensemble du SdS.

Dans ce contexte, il ne suffit pas de simplement adapter les méthodes et outils existants de développement. Il est nécessaire de proposer des concepts, des langages, des méthodes et des outils permettant la construction de SdS capables de s'adapter dynamiquement pour continuer d'assurer leur mission malgré les évolutions internes ou externes qu'ils subissent. Des cycles de vie itératifs et flexibles doivent être envisagés (Chiprianov *et al.,* 2014 ; LIUPPA & IRISA, 2014).

De plus, pour gérer l'indépendance opérationnelle des systèmes constituants, il faut être capable de définir une architecture et de construire un SdS qui pourra remplir sa propre mission, sans violer l'indépendance de ses systèmes constituants. Face à ce défi, il est envisagé de s'attaquer, d'une part, à la modélisation et l'analyse de propriétés non-fonctionnelles des SdS, en particulier aux défis liés à la sécurité et, d'autre part, à l'analyse, au développement et à l'évolution dynamique des architectures de SdS.

2.2. Gérer la diversité et la variabilité à grande échelle

La taille ou la composition d'un système (ou d'un SdS) n'est pas le seul facteur de complexité qu'il faut gérer. Un autre facteur concerne la diversité qui apparaît dans tous les domaines d'application et dans toutes les activités de développement. Cette diversité s'exprime dans les fonctionnalités à fournir à différents utilisateurs, dans les langages et les modèles à utiliser conjointement et dans les environnements d'exécution1, allant des réseaux de capteurs intelligents aux grandes infrastructures de calcul et de stockage de données. Plusieurs défis sont liés à cette problématique.

Concevoir un logiciel en fonction des besoins d'un client et l'adapter aux besoins d'un autre, en réutilisant une partie du code, ne peut plus se faire de façon artisanale. L'approche proposée par les lignes de produits logiciels permet de généraliser, et donc de maîtriser, la variabilité des logiciels.

Cependant, la gestion et la modélisation de la variabilité à grande échelle restent un défi. Dès lors qu'il faut gérer des milliers, voire des millions de configurations de produits logiciels, l'identification, la définition et la composition des variants se doivent d'être accompagnées de méthodes et d'outils permettant le passage à l'échelle (Metzger et Pohl, 2014).

Un autre enjeu réside dans la diversification et la spécialisation des langages utilisés par les acteurs impliqués dans la construction des logiciels. Ces langages sont spécifiques à un domaine (DSL ou Domain-Specific Language), permettant ainsi à un expert de développer rapidement, et de façon fiable, un logiciel adapté à son expertise. Actuellement, le développement de logiciels demande de nombreuses expertises. L'un des défis est de composer et coordonner ces langages de modélisation spécifiques de façon à construire des logiciels multi-domaines (Acher *et al.,* 2014).

2.3. Utiliser intensivement et facilement l'ingénierie dirigée par les modèles

Malgré de nombreuses avancées faites en Ingénierie Dirigée par les Modèles (IDM), son application dans bon nombre de domaines se limite à l'abstraction comme seule dimension de modélisation. Plusieurs problèmes cruciaux du point de

1. Le nombre de plates-formes Androïd distinctes est passé d'environ 11 000 à plus de 18 000 entre 2013 et 2014 (http://opensignal.com/reports/2014/android-fragmentation/).

vue de l'usage des modèles ont été identifiés (Whittle, 2014). L'un des problèmes concerne l'ergonomie et une part limitée des travaux et outils liés à l'IDM prend en compte les processus cognitifs généralement à l'œuvre lors de la conception et le développement. Le non-alignement est donc fréquent et entraîne généralement des processus cognitifs supplémentaires chez le concepteur pour adapter les modèles ou les outils (Bihanic *et al.,* 2014b). Si la syntaxe diagrammatique classiquement utilisée pour représenter les modèles répond pour partie aux attentes et visées fonctionnelles, il reste que celle-ci n'est pas suffisamment efficiente. La contribution du design peut s'avérer d'une aide précieuse en vue d'inventer et de concevoir de nouveaux langages de représentation et de visualisation graphique des modèles. Les applications courantes et futures nécessitent que la multiplicité des différentes sources de complexité soit maîtrisée en même temps (personnalisation et réactivité, accès aux non-informaticiens, prédictions et analyse, prise en compte des points de vue et des niveaux de compétences) (Bihanic *et al.,* 2013 ; Mussbacher *et al.,* 2014).

Partant de ce constat, un certain nombre de défis peuvent être identifiés en croisant les considérations sociétales et cognitives à plus ou moins longue échéance (Bihanic, 2014) :

1. Représentativité. Le caractère de plus en plus transdisciplinaire des applications et des systèmes doit nous inciter à, non seulement anticiper les difficultés techniques, mais aussi profiter de cette richesse. Cela exige aussi à repenser les formalismes de représentation, de description et de visualisation des modèles tout autant que les mécanismes utilisés en IDM pour les notations visuelles.

2. Accessibilité. Tandis que le logiciel est devenu réalité pour le grand public, l'IDM prône de placer les modèles au cœur des développements. C'est aux chercheurs dans le domaine de l'ingénierie dirigée par les modèles de rendre l'utilisation, la manipulation et la réalisation de modèles accessibles au plus grand nombre. Ceci implique aussi des modes d'interaction efficaces et intuitifs, qui exploitent au maximum les mécanismes visuels répertoriés (associativité, sélection, imposition, abstraction), et complémentés par des périphériques adaptés, c'est-à-dire flexibles et multi-dimensionnels.

3. Flexibilité. Les modèles doivent être à l'image des applications modernes. Ils seront ainsi plus faciles à utiliser et pourront être composés ou intégrés de différentes façons. Il s'agit de pouvoir tenir compte, dans un domaine d'application donné, et dans un contexte dynamique, de toutes les préoccupations, techniques ou relatives au domaine, avec plus de confiance et de prévisibilité dans le résultat d'une intégration. Ceci pourrait être facilité par l'utilisation de modèles de référence, facilement personnalisables et intégrables, ainsi que par des environnements adaptés lors des désynchronisations entre codes exécutables et modèles.

4. Évolutivité. Un modèle n'est jamais aussi riche que la réalité. Il est donc par définition incomplet, alors que les outils de modélisation se sont souvent intéressés au caractère complet et formel des modèles. Pour ne pas être un frein à la créativité, ces outils devraient au contraire permettre, par exemple, qu'un modèle ne soit pas conforme à son méta-modèle pendant la phase d'élaboration. Ainsi les aspects cognitifs liés aux utilisateurs devraient être pris en compte dans ces outils dès leur

création tout en ayant un impact sur les aspects techniques de maintenance, partage et évolution.

L'augmentation croissante de la complexité et de la diversité des systèmes, ainsi que leur regroupement en systèmes de systèmes sont des problèmes que notre communauté scientifique se doit de résoudre. Les chercheurs vont proposer de nouvelles approches pour favoriser l'adaptation dynamique, la sécurité, la composition, mais aussi maîtriser la variabilité à grande échelle et fournir des modèles ouverts, évolutifs et plus accessibles.

3. De la compilation à l'exécution des logiciels

La plupart des systèmes informatiques, des smartphones jusqu'aux accélérateurs de calcul (*i.e.*, GPU, FPGA, super-calculateurs des futurs centres de calcul) sont désormais pourvus de systèmes multi-cœurs. Ces nouveaux matériels mettent le parallélisme à la portée de tous, mais restent d'une grande complexité. Le défi est double. Il faut d'une part maîtriser la difficulté de programmation de ces architectures tout en assurant une certaine portabilité des codes et des performances. D'autre part, il faut offrir une meilleure interaction entre les utilisateurs et les compilateurs. Cela nécessite de revisiter les langages, notamment parallèles, les techniques de compilation (analyse et optimisation de codes), les systèmes d'exploitation (OS), mais également les moyens de construire les logiciels avec des techniques avancées du génie logiciel, tout en respectant des contraintes liées à l'espace, au temps, mais de façon plus récente à la consommation d'énergie.

3.1. Maîtriser la difficulté de programmation des architectures multi-cœurs

Dans ce cadre, la communauté compilation et calcul haute performance a identifié un ensemble de défis (Brandner *et al.,* 2014).

1. Les langages parallèles. Avec la mise sur le marché de plates-formes de plus en plus hétérogènes, le gain en performance des applications se fait maintenant en utilisant des processeurs dédiés à des tâches spécifiques. La programmation de tels systèmes logiciels/matériels devient beaucoup plus complexe, et les langages parallèles existants (MPI, openMP) semblent peu adaptés à cette tâche. Pour diminuer le coût du développement et du déploiement de logiciels sur des plates-formes spécifiques toujours changeantes, nous avons besoin d'approches et de langages qui permettent l'expression du parallélisme potentiel des applications et la compilation vers une plate-forme parallèle spécifique.

De plus, la complexité et la diversité de ces plates-formes justifient le développement de langages ou approches de haut niveau (comme X10, Chapel, OpenAcc, CAF, ou encore OpenStream) et d'optimisations au niveau source, en amont des langages natifs ciblés, dialectes souvent encore proches du langage C. Ceci offrira plus de portabilité de performances, mais aussi, pour l'utilisateur, une meilleure compréhension des transformations effectuées par le compilateur. L'amélioration de l'interface entre le programmeur et le compilateur est une

perspective fondamentale à améliorer. L'interaction langage/compilateur/ OS/environnement d'exécution est également un point clé à améliorer, la portabilité de la performance passant par l'adaptation à la plate-forme via cette interaction.

2. La compilation optimisante pour les performances en temps et en mémoire. La consommation d'énergie des systèmes (embarqués surtout) est maintenant en grande partie due au mouvement et au stockage des données. Pour s'adapter à cette problématique, les données et leurs mouvements doivent être exposés au programmeur, et les compilateurs doivent prendre en compte la trace mémoire et les mouvements des données par exemple entre deux calculs qui s'effectuent sur des unités de calcul différentes. Des approches existent déjà en compilation, notamment en ce qui concerne la compilation vers FPGA, mais ce n'est que le début vers une prise en compte plus générique des contraintes de mémoire. Dans un sujet connexe, la recherche de solutions dont le pire cas peut être garanti (WCET) est rendue encore plus difficile par la présence de parallélisme. à l'heure actuelle, la compilation à performances prédictives reste un problème largement ouvert.

3. La validation théorique des analyses et optimisations. Les analyses de programme (séquentiel ou parallèle), les transformations de code, la génération de code pour des architectures précises, doivent être définies et prouvées précisément. Si accompagner les algorithmes de leur preuve de correction reste essentiel, des techniques comme la preuve assistée et la translation validation permettent maintenant de valider des compilateurs entiers (par exemple Compcert). Le challenge ici réside dans l'application plus générale des méthodes formelles (*cf.* section 4), que ce soit pour définir précisément des algorithmes ou pour valider des optimisations dans le cadre du parallélisme en particulier.

4. La mesure et la reproductivité des résultats. Contrairement aux autres sciences expérimentales comme par exemple les sciences naturelles, la branche expérimentale de l'informatique souffre d'un manque de principe scientifique : la reproductibilité et la vérification des résultats expérimentaux en informatique ne sont pas encore entrées dans nos habitudes. Dans le domaine de la compilation optimisante en particulier, lorsque des performances sont publiées, il est très rare que ces performances puissent être vérifiées ou observées par une partie tierce. Définir des méthodes expérimentales de validation statistique de résultats et des modèles présupposés (modèles de coût, modèles de programmation, abstractions) reste un défi majeur pour notre communauté, même si certaines conférences de notre domaine commencent à proposer des évaluations expérimentales[2].

Ces problématiques orientées compilation/langages sont aussi liées aux problématiques générales du génie logiciel que sont la complexité et l'hétérogénéité des logiciels. La spécificité ici est que la même complexité est à prendre en compte aussi bien côté matériel (plates-formes hétérogènes, accélérateurs matériels) que côté logiciel dès lors que l'on veut faire coopérer différents matériels et logiciels.

2. http://evaluate.inf.usi.ch/artifacts

3.2. S'inspirer du domaine génomique pour des pilotes de périphériques adaptables

L'Internet des objets est entravé par le fait que le développement de pilotes de périphérique reste une tâche complexe ainsi qu'une source d'erreurs et exige un haut niveau d'expertise, liée à la multiplicité des systèmes d'exploitation cibles et au dispositif visé. Il est nécessaire de proposer de nouvelles méthodologies qui amènent une rupture forte avec les méthodes actuelles de développement de pilotes de périphériques.

Le défi consiste alors à considérer une nouvelle méthodologie pour comprendre les pilotes de périphériques, inspirée par le domaine de la génomique (Lawall, et Muller, 2014). Plutôt que de se concentrer sur le comportement des entrées/sorties d'un dispositif, cette nouvelle méthodologie reposera sur l'étude du code existant du pilote de périphérique de manière à le faire muter vers une version adaptée à l'environnement cible. D'une part, cette méthodologie devrait permettre d'identifier les comportements de pilotes de périphériques réels, de mettre à disposition ou non les caractéristiques du dispositif et du système d'exploitation, et d'améliorer les propriétés telles que la sécurité ou la performance. D'autre part, cette méthodologie a pour objectif de capter les patrons actuels de code utilisé pour réaliser des comportements, ceci en élevant le niveau d'abstraction à partir d'opérations individuelles vers des collections d'opérations réalisant une seule fonctionnalité, appelées gènes. Parce que les exigences du pilote sont figées, quel que soit le système d'exploitation, les gènes ayant des comportements communs dans des OS différents devraient pouvoir être mutualités, même lorsque leur structure interne diffère. Cela devrait conduire à des pilotes de périphériques réalisés par une composition de gènes, ouvrant ainsi la porte à de nouvelles méthodes pour résoudre les problèmes de pilotes lors de développement ou lors du portage de pilotes existants vers d'autres systèmes d'exploitation.

La nouveauté de ce défi réside dans l'élévation du niveau d'abstraction de la compréhension du code des pilotes de périphérique à partir des opérations individuelles sur des gènes. La description et l'analyse des codes en termes de gènes fournissent un cadre pour le raisonnement sur les opérations connexes. En outre, les spécifications utilisées par les outils de traitement de code peuvent devenir plus portables et adaptables lorsqu'elles sont exprimées en termes de gènes, permettant à une seule spécification d'être appliquée de façon transparente à des variants d'un gène unique, quel que soit le code réel cible.

Les différentes étapes nécessaires pour réaliser ce défi concernant le développement de pilotes périphériques par composition de gènes sont les suivantes :

1. Identifier les gènes en interaction avec le système d'exploitation, dans un premier temps à la main puis automatiquement. De tels gènes impliquent généralement des fonctions API OS, et ont une structure commune.

2. Identifier les gènes impliquant une interaction avec l'appareil dans un premier temps à la main puis automatiquement. De tels gènes impliquent généralement des opérations de bas niveau, qui sont spécifiques à chaque appareil.

3. Développer des techniques pour la composition de gènes en vue de construire de nouveaux pilotes de périphérique.

Un autre problème, déjà partiellement abordé dans la section précédente, concerne l'hétérogénéité des plates-formes considérées - des capteurs intelligents aux grandes infrastructures de calcul - et les formes multiples des fautes et pannes à considérer. Le défi est alors double : il faut fournir les abstractions pour modéliser, déployer et adapter de manière homogène les différentes couches d'architectures logicielles et les besoins multiples, tout en revisitant les problématiques de tolérance aux pannes et de sécurité.

3.3. Optimiser la consommation d'énergie lors d'exécutions sur architectures multi-cœurs

Un dernier défi concerne l'omniprésence des logiciels dans notre vie quotidienne. Ceux-ci engendrent une part importante de la consommation énergétique globale française (13 %). La communauté du génie logiciel peut jouer un rôle important dans la diminution significative et durable de cette consommation. En particulier, la consommation des logiciels s'exécutant sur des architectures multicœurs, qui sont aujourd'hui omniprésentes, que ce soit dans les serveurs ou les téléphones portables, demeure mal exploitée à ce jour : une part importante de l'énergie consommée est utilisée à mauvais escient.

Il est donc nécessaire de pouvoir comprendre finement comment les logiciels pilotent la consommation et d'identifier quels leviers peuvent être exploités dans les différentes couches d'un système informatique pour optimiser cette consommation en continu. L'éco-conception des logiciels apparaît donc aujourd'hui comme un défi crucial pour l'informatique afin de sensibiliser et guider les développeurs dans la réalisation de logiciels efficients. Cependant des gains notables ne peuvent être obtenus qu'en considérant des optimisations tout au long du cycle de vie du logiciel, depuis le recueil des besoins, durant leur exécution et jusqu'à leur maintenance, voire même leur recyclage.

La prise en compte des nouvelles caractéristiques matérielles pour construire les compilateurs, les pilotes de périphériques ou encore réduire la consommation d'énergie des logiciels semble évidente. Ce constat amène de nombreuses questions auxquelles les chercheurs de ces domaines devront apporter des réponses, aussi bien en termes de nouvelles structurations des logiciels qu'en termes d'optimisation de ces structures.

4. Méthodes et outils de validation et de vérification

Les nouveaux domaines d'application (informatique embarquée, intelligence ambiante, Internet des objets) et les nouvelles architectures (informatique dans les nuages, Software as a Service ou système de systèmes) font émerger de nouvelles propriétés ou contraintes (architectures reconfigurables, dynamicité de l'environnement, évolution des usages), en plus des propriétés classiques, telles que la sûreté et la sécurité. Un défi majeur consiste à revoir les méthodes de validation et de vérification (V&V) pour prendre en compte ces nouvelles architectures et les nouvelles propriétés associées.

4.1. Créer des méthodes évolutives

Une des difficultés est que les techniques actuelles de V&V requièrent la connaissance de la logique applicative pour garantir la qualité avant déploiement. Pour ce faire, elles s'appuient sur des spécifications de référence, qui n'anticipent pas a priori les prochaines évolutions. Dans ces conditions, il faut garantir la qualité de systèmes, qui vont devoir s'adapter à des univers flexibles et ouverts, et dont on ne peut prévoir les nouvelles fonctionnalités, les nouveaux usages, et les évolutions environnementales. Une des pistes peut être de s'appuyer sur un apprentissage de l'évolution incluant un processus de validation dynamique (pendant le déploiement), par exemple sur la forme de test passif ou de monitoring (Dadeau, 2014).

4.2. Mieux intégrer les méthodes dans le monde industriel

Au-delà de la prise en charge de ces nouvelles propriétés, un défi récurrent pour la V&V est de faire pénétrer plus amplement les approches et outils dans le monde industriel.

Dans le cadre de la vérification de modèles (*model-checking*), l'impact dans l'industrie est, à ce jour, principalement limité aux systèmes embarqués critiques. Deux raisons principales sont d'une part, la réponse binaire à des propriétés de satisfaction qui n'est pas suffisamment informative, et d'autre part l'abstraction insuffisante pour répondre au réglage et à l'évolutivité des systèmes. Il faudra surmonter ces limitations, par exemple, en offrant des méthodes formelles paramétriques pour la vérification et l'analyse automatisée du comportement des systèmes. L'enjeu est d'obtenir des garanties sur la qualité des systèmes en fonctionnement, dès la phase de conception (André *et al.,* 2014).

Parvenir à une meilleure automatisation des preuves déductives est elle-aussi essentielle, mais ce n'est pas le seul levier. Réutiliser spécifications et preuves associées serait un atout important. Modularité, héritage, paramétrisation facilitent la réutilisation, mais ne suffisent pas. Des approches inspirées des lignes de produits logiciels ou des approches à la carte ont été proposées, mais ces mécanismes demandent à être simplifiés. Un autre mode de réutilisation serait possible en développant un standard permettant l'interopérabilité entre les différents outils et

formalismes, permettant ainsi la réutilisation de preuves dans un formalisme donné (*in the small*) ainsi que la réutilisation des preuves dans différents formalismes (*in the large*) (Dowek et Dubois, 2014).

Le test à partir de modèles ou *model-based testing* (MBT) représente le moyen principal pour automatiser la génération et l'exécution de tests fonctionnels (Utting et Legeard, 2007). En plus de permettre la traçabilité des exigences jusqu'aux tests, cette approche fournit des métriques d'avancement du processus de validation. Mais l'adoption des approches MBT dans l'industrie se heurte au problème de la conception de modèles. Les travaux sur l'inférence automatique de modèles ont montré leur intérêt et doivent être poursuivis. Par ailleurs, l'utilisation massive des approches MBT dans le monde industriel passe par un accompagnement des équipes de validation, et la construction de formalismes et outils adaptés (Dadeau et Waeselynck, 2014)

4.3. Améliorer le passage à l'échelle des méthodes

Le passage à l'échelle est aussi un enjeu majeur pour l'adoption des différentes approches de V&V dans le monde industriel, notamment pour faire face à l'augmentation de la taille des systèmes. La capacité d'exprimer de grands modèles avec une représentation et des outils adaptés est un défi qui rejoint les problématiques déjà présentées section 2. Cependant le passage à l'échelle ne se limite pas à la seule expression de la spécification sous la forme d'un modèle. Par exemple, avec le recours aux techniques de développement agiles, les tests passent au premier plan, parfois construits avant le code, et ré-exécutés à chaque mise à jour. L'augmentation de la taille du code (et donc du nombre de tests) rend problématique la ré-exécution systématique de tous les tests. La priorisation des tests devient essentielle. Dans ce contexte où l'application est amenée à évoluer, la maintenance de nombreux tests est un défi à part entière avec par exemple l'invalidation des tests devenus obsolètes ou encore la mise à jour des tests encore pertinents.

4.4. Améliorer la gestion des fautes

La détection de fautes ou de défaillances (bug) est l'objectif premier des techniques de V&V. Dans la plupart des domaines, la complexité et la richesse des spécifications, des utilisations et des environnements d'exécution rendent caduc le désir d'un logiciel sans faute. Le but de l'ingénieur face à ce fait n'est donc pas d'éradiquer toutes les fautes mais de gérer au mieux celles qui demeurent.

Les pionniers Avizienis *et al.* (2001) ont défini quatre axes principaux dans la gestion des fautes : la prévention, la tolérance, l'estimation, et la suppression des fautes. À l'heure actuelle, les approches proposées concernent des fautes simples, les fautes difficiles étant laissées à l'expertise et l'intelligence humaine. Pour 2025, un défi consiste à envisager ces quatre tâches relatives à la gestion de faute comme des tâches d'intelligence artificielle, où, pour un type de faute donné, la machine devient de façon comparable aussi bonne que l'expert humain.

Ainsi, l'étape ultime de la suppression de faute est une réparation automatique (Goues *et al,* 2013). Un logiciel intelligent analyse un rapport de défaillance, identifie la faute et corrige automatiquement le code source. Attaquer ce défi nécessite une approche interdisciplinaire. Premièrement, des experts en logiciel (génie logiciel, programmation, système) doivent collaborer avec des experts en décision (apprentissage, fouille de données, contraintes) (Xie, 2009). Deuxièmement, une approche interdisciplinaire de la résilience des systèmes avec des chercheurs en biologie, écologie, physique et théorie des systèmes devrait faire sauter des verrous conceptuels dus au cloisonnement actuel.

Face à de nouveaux domaines et de nouvelles architectures logicielles, les méthodes de V&V se doivent d'évoluer et de mieux pénétrer le monde industriel. Les chercheurs du domaine s'attaquent à cette problématique en améliorant les techniques actuelles de réutilisation, d'automatisation en prenant en compte le passage à l'échelle, tout en les faisant évoluer. La détection de fautes et leur correction seront également revisitées en s'appuyant largement sur des approches interdisciplinaires.

5. Conclusion

Les trois challenges décrits par Sommerville (2006) comme étant les défis du 21e siècle pour le domaine du génie logiciel et de la programmation restent d'actualité :

1) l'héritage de logiciels existant depuis de nombreuses années et devant continuer à fonctionner et à évoluer,

2) la prise en compte de l'hétérogénéité des matériels et des systèmes d'exploitation sur lesquels s'exécutent les logiciels qui doivent être construits de façon souple pour être adaptables à ces environnements, tout en étant fiables, et finalement

3) la diminution des délais de livraison des logiciels, traditionnellement longs, sans compromettre la qualité du système.

D'autres documents récents font également état de défis dans le domaine du génie logiciel et de la programmation. On peut citer le NESSI white paper intitulé *Software Engineering Key Enabler for Innovation* (NESSI, 2014) et le papier de la commission européenne écrit par ISTAG et intitulé *Software Technologies, The Missing Key Enabling Technology* (ISTAG, 2012). Dans ces deux documents, nous retrouvons les défis précédemment cités, mais également l'importance de la coopération des académiques avec les industries du logiciel, le fait que des dépôts de logiciels open source soient créés et maintenus, et qu'un enseignement de qualité en génie logiciel et programmation soit assuré dans les universités et écoles d'ingénieurs.

Les trois grandes questions qui ont permis de classer les différents défis proposés par les chercheurs du GDR GPL reposent, quant à elles, sur les différentes étapes de construction des logiciels, allant de l'expression des besoins jusqu'à l'exécution et

au-delà par une réflexion sur l'évolution et la maintenance du logiciel. L'importance des nouveaux domaines d'application, mais également l'évolution des architectures matérielles obligent à revoir ces étapes, en y intégrant les nouvelles propriétés liées à ces nouveaux contextes. La nécessité de construire ces logiciels de façon fiable, mais également en prenant en compte la dimension du temps de mise sur le marché oblige les chercheurs à revoir les méthodes et outils pour aller plus rapidement vers des logiciels ajustés au mieux à leur contexte, tout en étant adaptables.

Remerciements

Les présentations et les échanges lors de la table ronde sur les défis 2025 des Journées Nationales du GDR GPL à Paris en Juin 2014 et la journée sur les défis 2025 en septembre 2014 à Paris ont permis de rédiger cette synthèse. Les auteurs remercient l'ensemble de leurs collègues ayant participé à ces événements et ayant rédigé des contributions, avec un remerciement particulier à Catherine Dubois pour sa relecture attentive.

Bibliographie

Acher M., Barais O., Baudry B., Blouin A., Bourcier J., Combemale B., Jézéquel J-M., Plouzeau N. (2014). Software diversity : Challenges to handle the imposed, opportunities to harness the chosen. In (Dubois *et al.,* 2014).

André E., Delahaye B., Habermehl P., Jard C., Lime D., Petrucci L., Roux O. H., Touili T. (2014). Beyond model checking : Parameters everywhere. In (Dubois *et al.,* 2014).

Avizienis A., Laprie J-C, Randell B, et al. (2001). *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science.

Bihanic D., Bruel J-M, Collet P., Combemale B, Dupuy-Chessa S., Le Pallec X, Polacsek T. (2014). L'IDM de demain : un accès facilité, un usage intensif pour des performances accrues. http://gdr-gpl.cnrs.fr/node/160

Bihanic D., Chevalier M., Dupuy-Chessa S., Morineau T., Polacsek T., Le Pallec X., et al (2013). Modélisation graphique des SI : Du traitement visuel de modèles complexes. In *Inforsid 2013*.

Bihanic D., Dupuy-Chessa S., Le Pallec X, Polacsek T. (2014). Manipulation et visualisation de modèles complexes. In (Dubois *et al.,* 2014).

Brandner F., Cohen A., Darte A, Feautrier P., Fursin G., Gonnord L., Touati S. (2014). Défis en compilation, horizon 2025, http://gdr-gpl.cnrs.fr/node/160.

Chiprianov V., Gallon L., Munier M., Aniorte P., Lalanne V. (2014). The systems-of-systems challenge in security engineering. In (Dubois *et al.,* 2014).

Dadeau F. Waeselynck H. (2014). Les défis du test logiciel - bilan et perspectives. In (Dubois *et al.,* 2014).

Dowek G., Dubois C. (2014). Réutiliser les spécifications et les preuves, http://gdr-gpl.cnrs.fr/node/160.

Dubois C., Duchien L, Levy N., editors. (2014). *Actes des Sixièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, France, Juin 2014. Conservatoire National des Arts et Métiers. https://hal.inria.fr/hal-01055907/en

Duchien L., Ledru Y. (2012). Défis pour le Génie de la Programmation et du Logiciel GDR CNRS GPL. *Technique et Science Informatiques* (TSI), 31(3) :397–413.

Goues C., Forrest S., Weimer W. (2013). Current challenges in automatic software repair. *Software Quality Control*, 21(3) :421–443.

ISTAG (2012). Software technologies, the missing key enabling technology- digital agenda for Europe. http://cordis.europa.eu/fp7/ict/docs/istag-soft-tech-wgreport2012.pdf

LIUPPA & IRISA (2014). Défis dans l'ingénierie logiciel des Systèmes de Systèmes, http://gdr-gpl.cnrs.fr/node/160.

Lawall J., Muller G. (2014). The future depends on the low-level stuff. In (Dubois *et al.,* 2014).

Metzger A., Pohl K. (2014). Software product line engineering and variability management: achievements and challenges. *In Proceedings of the on Future of Software Engineering*, pages 70–84. ACM.

Mussbacher G., Amyot D., Breu R., Bruel J-M, Cheng B., Collet P., Combemale B., France R., Heldal R., Hill J., et al. (2014). The relevance of model-driven engineering thirty years from now. Dans les actes de *Model-Driven Engineering Languages and Systems*, pages 183–200. Springer International Publishing, 2014.

NESSI (2014), Networked European Software NESSI White Paper and Services Initiative. Software engineering key enabler for innovation. http://www.nessi-europe.eu/Files/Private/NESSI_SE_WhitePaper-FINAL.pdf

Sommerville I. (2006). *Software Engineering* : (Update) (8th Edition) (International Computer Science). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Utting M., Legeard B. (2007). *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Whittle J., Hutchinson J., Rouncefield M. (2014). The state of practice in model-driven engineering. *Software, IEEE*, 31(3) :79–85.

Xie T., Thummalapenta S., Lo D., Liu C. (2009). Data mining for software engineering. *Computer*, 42(8), 55–62.